by Peter Collinson, **Hillside Systems**

BEK SHAKIROV

Kernels and Files

One of the perils of publishing is the time lag between the day I cause the keys on my keyboard to move up and down creating words and the time you hold the magazine in your hands casting your eyes across the pages. I started last month's article with a quiet reflection on the mildness of the change from summer to winter in the UK, which was true as I was writing. However, winter came in with a bang; we had such violent storms, gales and nationwide flooding that it was noted in the U.S. media and I've had several "Are you OK?" bits of email from friends across the pond. Of course, by the time you read this, the news story will be long gone from public consciousness.

Another publishing peril with a long running column like this is the feeling that "I've written about that." I often get that feeling when I am thinking up a topic. However, I then look at past articles and find that the subject was actually covered sufficiently long ago but it seems reasonable to take a fresh look. This is the case with this month's article. I've certainly written on the topic of how applications interact with files and directories before, with different slants, but the topic is so crucial to the operation of UNIX that it's worth taking a new look.

I suppose that the most basic of the "UNIX Basics" is the notion of the file system. UNIX was created by Ken Thompson in Bell Labs to experiment with ideas of file system design. At one time, the largest part of the UNIX kernel, the memory resident part of the operating system, was concerned with handling the files on the disk (or disks) and providing user applications with a set of services that enable them to deal with files. I'm sure that in code size terms, this part of the kernel has been outstripped by the sections that deal with networking.

I am going to look in more detail at how applications use files and how the operating system kernel works to provide these services to applications. On UNIX, we call an application that's running a *process*. Many processes will run on a live UNIX system, and all will use the memory-resident kernel to provide system services. One of the main jobs of the kernel is to manage the set of processes running on the machine at any one time. The kernel ensures each process is given some CPU cycles when it needs it, and arranges for each process to be resident in memory so it can run when it needs to.

The kernel provides a standard set of services that allow processes to interact with it, and also acts to simplify the task of the application writer. In UNIX, we tend to call these services "system calls," because the application calls a routine in the system (i.e., the kernel) to undertake some specific task. At a programming level, a system call is indistinguishable from any call to any other routine. However, for a system call, code in the kernel will perform the task for the process. It returns a result to the process saying that the job has been done, or perhaps delivering requested information. As time has gone on, the set of system calls has expanded to add new functionality to the

interactions between applications and the kernel. Also, the tasks of some system calls have been taken over by others, and the UNIX system provides an interface routine to ensure that programs are backwards compatible. Section 2 of the online manual on most systems describes the set of system calls supported by your system.

I often like to say system calls provide the application with a model of how it needs to interact with the outside world and also supply a set of simplifications of how the outside world operates. The kernel fills in the gaps between what the application is expecting to happen and how the hardware needs to be controlled.

Opening Files

When saving information in a file, for example, a process wants to open a file by name, write some data and close the file, knowing later that it (or some other process) can open the file by quoting its name and then read the data back. We humans prefer to give names to files, and if we are sensible, tend to use the name of a file to indicate something about its contents. When we write applications, we're happy not to have to worry about the precise location in the hardware of where the file is stored on the disk or how the file is actually mapped onto specific disk blocks. System calls provide a layer that simplifies the task of managing files from the application.

What does the kernel do for a program that's just dealing with I/O from or to the disk? It examines the name the process gives it and negotiates the file system hierarchy to find the directory where the file lives or will live. This action is the task of the open system call that the application employs to open an existing file or create a new one. The open call returns a small positive number used in any subsequent system calls to refer to the file that has been opened or created.

We've started to break down the I/O needs of the application into a small set of primary tasks, and at this stage the action of opening a file implies that the kernel retains some information about each of the files each process has opened. Actually, it's possible for several processes to have opened the same file, and be at different stages of reading or writing data to or from that file. The kernel manages this need by retaining a small block of information for each file a process opens. The information block contains a pointer into the file that is the offset where the next I/O operation on that file will occur for that particular process. When several processes open the same file, each will have their own pointer into the file. It's also possible for a single process to open a file more than once and maintain several points of access at different offsets.

About three paragraphs back, I skipped rather quickly over the other primary function of the open system call: translating a file name supplied by the application into a specific file on the disk. All operating systems need a method of identifying a specific file on the disk by name to enable a process to read the data it contains, or will provide a mechanism to create a new file tagging it with a name.

Hang on. What *is* a file on a disk? Let's start by thinking about a more fundamental question: What is a disk? This question should perhaps be expressed as: How does the kernel see a disk? What model of operation does a disk controller present to the kernel?

Disks comprise sets of blocks each storing some fixed number of bytes, and for a long time, the default size of a block has been 512 bytes. Each block is numbered so that it can be addressed separately. The controller can be given a block address and told "go to that block and read some number of blocks into memory." Or, for writing, told "go to that block and write some number of blocks from memory to the disk." The controller can only deal in complete blocks, so if the application wants to overwrite the first byte of a block, the kernel needs to pull that block into memory, make the change, then write the complete block out again.

The process of moving data to and from the disk usually happens without direct intervention of the CPU that runs the machine. The disk controller uses Direct Memory Access (DMA) techniques to move data between the computer's memory and the disk surface. It's told to move some number of blocks and will do just that, only worrying the CPU when the operation is over and the controller needs something else to do.

This model has remained constant for many years, and the 512 byte block has become a standard. What has changed is the total number of blocks a disk can hold has grown significantly while the physical size of the disk drive has shrunk from the size of a large washing machine to smaller than a paperback book.

However, notice that the kernel doesn't have to deal with individual 512 byte blocks; it can happily work with a basic transfer unit that comprises several blocks. The reality is that transferring more than one block is only a little bit slower than moving just one. The cost of managing disks is the time it takes to move the heads to the correct position to start the I/O operation in the first place.

System designers are presented with a trade-off. It's faster to make the hardware treat the disk as if it had, say blocks of 8 K, which we call a "logical" block. Each disk operation the kernel initiates will ask the hardware to move 16 "physical" 512-byte blocks in one operation. However, by doing this, we waste space on the disk. A file that contains a single byte must occupy at least one logical block, and so 8,191 bytes of the disk will be wasted. Kirk McKusick's fast file system forms the basis of the Solaris disk system and uses a mixture of block sizes attempting to balance speed of data transfer against disk wastage. Most Solaris file systems seem to use a basic logical block size of 8 K, but the system allows for 1 K "fragments" intended to reduce space wastage.

Storing Files

OK, then, the disk model the kernel sees is a set of numbered blocks, and it may choose to create larger logical block sizes if it wishes, but we'll forget about that distinction now. It can access each block randomly, so it can find a specific block relatively quickly. What next? Well, this model is some way away from what we want to present the process. The kernel needs to support a method of organizing the files on the disk, and we call this a *file system*. A file system will contain some method of storing file names and other information about each file, often called the file's *metadata*. It will also need a method to manage the free space on the disk, so it knows which parts are empty and which are full. A file will be a collection of blocks, and the file system needs to know where each block

of a file is on the disk, so it can return the file in the correct order when an application reads its contents.

The earliest type of file systems employed a simple method of storing a file. Each file is written into a set of contiguous blocks along the disk, so all the file system has to remember to find a file is the address of the first block of the file and the number of blocks the file occupies. This method of file storage is known as "base and extent" and was used on the early "big iron" operating systems. Actually, it was also used in MS-DOS for the FAT file system on floppies and is still used on Windows systems today.

The base and extent storage system works well for single-user systems where only one process is writing a single file to the disk at any moment in time. But, there are problems. File deletion leaves holes, and these holes are hard to fill. The `defrag` program is seen to be the solution to these problems for Windows. These problems would be compounded if we considered using the mechanism on a multiuser system where several processes can be writing to several different files at once. UNIX supports a more complex file system structure that makes creating files somewhat easier, allowing efficient handling of the free space on the disk. But, it should be said, at the cost of making file reading somewhat less efficient.

UNIX splits its disks into two chunks. Don't get confused here. UNIX typically splits disks into partitions (Sun calls them slices) each of which contain a separate file system. I am talking now about one of these partitions, which can encompass the entire physical disk or may be just a portion of it. A partition is really a "logical disk" holding a file system. Getting back to the plot …

UNIX splits its disk into two chunks: one chunk contains *inodes* or "index nodes;" the other contains the data and the free blocks on the disk. The early UNIX system had two separate areas for the two chunks; more modern systems interleave inode and data areas across the disk, splitting the disk into what are known as "cylinder groups."

The *inode* describes a file. It contains the metadata, like ownership, permission information, various times and most importantly, file size. It also holds the block addresses of the first few blocks in the file. On Solaris, the inode addresses 12 blocks directly, so small files of up to 12 logical blocks can be read by pulling the inode into memory and just reading the blocks to which the inode contents refer. To cater for larger files, the inode contains "indirect pointers;" three on Solaris. Each of these indirect pointers address a block on the disk, which in turn contains a set of block addresses.

Essentially, the file is stored in a linked data structure. To read data from the file, first pull in the inode that describes the file, then use the information it contains to read blocks that can be randomly scattered all over the disk. When we append to the end of the file, we need to obtain unused disk blocks to place data into them, and then update the inode with the new block contents. When the file grows and jumps from a small file (less than 12 directly accessed logical blocks) to a large one, we now have to maintain an indirect block table in an allocated block.

In the original UNIX systems, the free blocks on the disk were linked together in a list. The kernel maintained a table of the next hundred free blocks on the disk. When they were exhausted, it scanned the disk to find another hundred free blocks, which was often time-consuming. Modern systems use bitmaps to indicate free blocks and attempt to maintain a locality of reference, so the inode and its associated blocks are placed physically close to each other, minimizing seek times for file access.

Names

You may have spotted that the inode for a file doesn't contain its name. Each inode on a disk has a unique number associated with it, and a file is really referenced by an inode number and not a name. We need some entity in the file system that maps names onto inode numbers, and this mapping is done by the directory structure.

Directories are just normal files. Their contents are stored on the disk using the same linked structure described above and are addressed like a file using an inode number. Directories hold a set of records that map names to inode numbers. There is only one piece of magic (the root directory of a particular file system is always found at inode no. 2). Incidentally, it's inode no. 2 and not inode 1 for historical reasons. Inode 1 was used on UNIX Version 6, but the root directory moved to inode 2 when UNIX Version 7 was released. Although the intention was originally to use inode 1 to hold bad blocks on the disk outside the file system hierarchy, this intention was superseded by events.

One side effect of the use of directories to map names onto inodes is the ability to generate file *links*. It's easy to create a new directory entry that maps a new name onto an existing inode number, so we can create several directory entries that point at the same physical file contents. We need to add a reference count to the inode telling the system how many directory entries point at it. This is done to aid file deletion. When an application deletes a file, the kernel removes the directory entry, but the actual file contents can't be removed until the number of links drops to zero. Notice also that inodes are only unique in a specific disk partition, so links can't span across different file systems. The symbolic (or soft) link, which is a file containing the name of a file, was developed to get around this limitation.

Putting Things Together

When the user types say:

```
$ cat /etc/motd
```

the `cat` command attempts to open the named file by using the `open` system call. The kernel starts looking in the disk partition known to hold the root file system and because the desired pathname starts with a `/`, it reads the directory found at inode 2. This file is pulled into memory and scanned looking for `etc`. Eventually, the name will be found, and an inode number obtained for it (No. 421632 on my system). Inode 421632 is read, and its blocks pulled into memory to be scanned for the string `motd`. Once the file is found (inode 421720 on my system), the job of the `open` system call is over. The open file table for the running `cat` command will be updated to refer to inode No. 421720 and the file position pointer set to zero. The kernel will return control to the `cat` command, saying "yes, I opened that" and return the small number needed to deal further with that file.

The `cat` command will issue a `read` system call to obtain the contents of the file, or probably the first 1 K of the contents. The kernel will use the stored file pointer to access the appropriate place in the `motd` file, read the data, and return it to the `cat` process. Actually, the kernel code for the `open` system call will have anticipated this is what the process will do, and will have initiated the read of the first block of data without being told. A read-ahead of the next block speeds things up in most cases; the disk can be busy doing the read using DMA while the process is deciding what to do next, or processing the last block of data it received. The `cat` program will use the `write` system call to output the text in the first 1 K of data to the user's screen.

Actually, if you use `truss` to see what's really going on when a Solaris `cat` command is executed, you'll find something different is happening. The `cat` command uses memory mapping to obtain the contents of the file. However, what I described above works, and would be the way that a mortal would write the `cat` command.

What the Kernel Does

We started with a scenario where we wanted to allow an application to be able to issue some system calls to open a file, and then read or write some information from or to a file. We can see the kernel engages in considerable work to allow this simple set of actions to be mapped onto some instructions to move the disk heads to the right place on the disk to store and retrieve the data.

The kernel needs to understand how to read files in order to read directory contents, how the directory contents map onto inode numbers, and how a file position on a particular inode is mapped into a specific block on the disk. It will undertake a set of checks to ensure the user has rights to do what they're trying to do. When creating a new file, it needs to know how free blocks are managed, and how it can obtain an unused block to store data. Actually, before it gets to this stage, it needs to know how to obtain an unused inode for the new file, and will need to initialize the inode to contain the correct metadata. It will also need to check in the directory that the intended filename is unique.

It does all this in a multiuser environment, where at all times it deals with what are essentially random requests for services. The kernel will also try to maintain economy of action. When it reads some information, it attempts to hang onto it so subsequent calls can use the copy stored in memory.

A disk inode on Solaris occupies 128 bytes, for example, but a disk read to obtain that inode will read a logical block from the disk, which is usually 8,192 bytes. So an attempt to read a single inode will also read another 63 inodes that weren't actually wanted at that time. The thinking here is there's a good chance the user process will open a file whose inode has already been read.

Actually, this doesn't work for the `/etc/motd` on my system. The inode number for `motd` is 88 inodes away from the inode number for `etc`, so it's found in the next block–at least it's in the next block and unlikely that a disk head movement is needed to read it. The kernel engages in a considerable amount of this type of information caching which sometimes pays off and sometimes doesn't.

The tale I've presented you with is simplified. There's quite a lot more that goes on in a live system to try to make actions happen quickly. There's also many other constraints and design decisions that I didn't have space to cover here.

You can find an exhaustive reference on the topic of UNIX kernels by getting "The Design and Implementation of the 4.4 Operating System" by Marshall Kirk McKusick, Keith Bostic, Michael Karels and John Quarterman (published by Addison Wesley and ISBN 0-201-54979-4). This is a revised edition of the 4.3BSD book.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.